HADDOCK: A Language and Architecture for Decision Diagram Compilation

R. Gentzel[1], L. Michel[1], and W.-J. van Hoeve[2][0000−0002−0023−753X]

[1] University of Connecticut, Storrs CT 06269, USA
[2] Carnegie Mellon University, Pittsburgh PA 15213, USA
rebecca.gentzel@uconn.edu,ldm@engr.uconn.edu,vanhoeve@andrew.cmu.edu

Abstract. Multi-valued decision diagrams (MDDs) were introduced into constraint programming over a decade ago as a powerful alternative to domain propagation. While effective MDD-propagation algorithms have been proposed for various constraints, to date no system exists that can generically compile and combine MDD propagation for arbitrary constraints. To fill this need, we introduce HADDOCK, a declarative language and architecture for MDD compilation. HADDOCK supports the specification, implementation, and composition of a broad range of MDD propagators that delivers the strength one expects from MDDs at a fraction of the development effort and with comparable performance. This paper describes the language, the framework architecture, outlines its performance credentials and demonstrates how to specify and implement novel MDD propagators.

1 Introduction

Binary decision diagrams (BDDs) were first introduced to represent Boolean functions in the context of switching circuit verification [27,1]. They became widely popular within various branches of computer science after Bryant proposed effective algorithms to compile BDDs with a fixed variable ordering [7,8]. Since then, many variants of decision diagrams have been developed, including multi-valued decision diagrams (MDDs) for representing functions with discrete (multi-valued) variables [30]. In the context of constraint programming, decision diagrams can be interpreted as a compact graphical representation of the solution set to a given discrete structure, typically represented by a (global) constraint. For example, decision diagrams were used to develop propagation algorithms for constraints over sets [25,20,13], n-ary table constraints [10], and REGULAR constraints [9]. All these works use MDDs as an efficient data structure to perform traditional domain propagation.

Andersen et al. [2] were the first to recognize that instead of propagating domains, it is possible to *propagate MDDs*: in addition to a domain store, the constraint solver maintains an MDD store on which constraints perform filtering. The key contribution of [2] is to introduce *relaxed* decision diagrams that are of polynomial size by explicitly limiting their width (the maximum number of nodes per layer). When the width is limited to one, MDD propagation defaults to

domain propagation. Larger widths can lead to increasingly more effective MDD propagation, as shown in a range of papers that study the compilation and application of MDD-based constraint programming [16,18,17,19,22,11,4,24].

MDD propagation methods are all based on the same underlying principles for compiling, refining, and filtering the decision diagram, as summarized in [22]. Yet, their implementations are often dedicated to their specific task or purpose [15,14,28,29,12]. To date, no system exists that allows users or developers to easily define and combine MDD propagators in constraint programming models. In this work, we take on this task and present HADDOCK,[3] a language and architecture that uses an MDD specification to automatically compile and integrate decision diagrams into a CP solver.

Contributions. Our primary contribution is the introduction of a *specification language* and associated implementation architecture that not only allows the automatic compilation of the diagram, but also generates the rules for refining (splitting) and filtering (propagating) MDD abstractions. This concretizes and generalizes the framework for MDD-based constraint programming proposed in [22]. While that framework allows one to describe MDD propagators based on transitions between states, it does not provide concrete functionality for generic propagation, refinement, or composition. Another related compilation framework is proposed in [6,5], which uses a dynamic programming formulation as input to compile the diagram. We will apply similar concepts, but we note that a dynamic programming model alone is not sufficient, e.g., for describing some of the MDD propagation rules or the integration into a CP solver. Instead, we adopt the formalism of *labeled transition systems* as an abstraction of MDDs.

We have implemented HADDOCK in the C++ version of MiniCP [26].[4] So far, HADDOCK contains MDD specifications for ALLDIFF, AMONG, GCC, SEQUENCE, and (weighted) SUM constraints, as well as some problem-specific MDD propagators. HADDOCK allows the user to declare multiple MDDs within a CP model, each associated with a suitable set of constraints, and automatically integrates the MDD propagators into the constraint propagation fixpoint computation. HADDOCK offers comparable performance for MDD propagation at a fraction of the development effort needed for dedicated implementations.

Motivating Example. We will use the AMONG constraint as the running example for this paper. Recall that $\text{AMONG}(x, lb, ub, S)$ is defined as

$$lb \leq \sum_{i=0}^{n-1} (x_i \in S) \leq ub \tag{1}$$

for an array of n variables x, a lower bound lb, an upper bound ub, and a set of values S. In [22], an MDD propagator for AMONG, establishing MDD consistency in polynomial time, was proposed and implemented. The pseudo-C++ HADDOCK fragment shown on the right of Fig. 1 generates code with

[3] HADDOCK stands for 'Handling Automatically Decision Diagrams Over Constraint Kernels'. It also refers to a saltwater fish, as well as to a fictional character from the Tintin comic series by Hergé, both of which are however irrelevant to this paper.

[4] Code: `https://bitbucket.org/ldmbouge/minicpp/commits/tag/HADDOCK`

```
int main() {
  int width = 64, H = 40;
  int L1 = 0, U1 = 6, N1 = 8;
  int L2 = 22, U2 = 30, N2 = 30;
  int L3 = 4, U3 = 5, N3 = 7;
  auto cp   = makeSolver();
  auto vars = boolVarArray(cp, H);
  auto mdd = new MDD(cp, width);
  for (int i=0; i<H-N1+1; i++)
    amongMDD(mdd,
          vars.sub(i,i+N1),L1,U1,{1});
  for (int i=0; i<H-N2+1; i++)
    amongMDD(mdd,
          vars.sub(i,i+N2),L2,U2,{1});
  for (int i=0; i<H-N3+1; i+=7)
    amongMDD(mdd,
          vars.sub(i,i+N3),L3,U3,{1});
  cp->post(mdd);
   ...
}
```

```
void amongMDD(MDDSpec& mdd,
              vector x,int lb,int ub,
              set<int> S)   {
  mdd.add(x);
  auto c = mdd.makeConstraint(x,"amongMDD");
  int Ld = mdd.addState(c,0),Ud = mdd.addState(c,0);
  int Lu = mdd.addState(c,0),Uu = mdd.addState(c,0);
  mdd.arcExist(cs,[=](p, c, x, v) -> bool {
      return (p[Ld] + v∈S + c[Lu] <= ub) &&
             (p[Ud] + v∈S + c[Uu] >= lb);
   });
  mdd.forward(Ld,[=](o,p,x,v){o[Ld]=p[Ld]+v∈S;});
  mdd.forward(Ud,[=](o,p,x,v){o[Ud]=p[Ud]+v∈S;});
  mdd.reverse(Lu,[=](o,c,x,v){o[Lu]=c[Lu]+v∈S;});
  mdd.reverse(Uu,[=](o,c,x,v){o[Uu]=c[Uu]+v∈S;});
  mdd.relax(Ld,[=](o,l,r){ o[Ld]=min(l[Ld],r[Ld]);});
  mdd.relax(Ud,[=](o,l,r){ o[Ud]=max(l[Ud],r[Ud]);});
  mdd.relax(Lu,[=](o,l,r){ o[Lu]=min(l[Lu],r[Lu]);});
  mdd.relax(Uu,[=](o,l,r){ o[Uu]=max(l[Uu],r[Uu]);});
}
```

Fig. 1. Pseudo-C++ code to create a nurse rostering model using an MDD (left) and pseudo-C++ HADDOCK code for creating an AMONG MDD propagator (right).

equivalent MDD propagation behavior. The `amongMDD` function takes a reference to an `MDDSpec` object *mdd* that *accumulates* all the specifications. Its other arguments are the array of variables x, the bounds lb and ub, and the set of values S. Line 4 tracks the array of variables and line 5 creates a descriptor for the four properties specified in lines 6-7 and to be held in a state. The remainder of the code relies on closures to define the arc existence condition as well as the transition functions and relaxations for the four properties. The method `forward` is used to add top-down transitions while `reverse` adds bottom-up transitions.

With 18 lines of code, a developer specifies a *reusable* factory that models instances of AMONG. Multiple calls to this factory results in a *composition* of specifications to model their conjunction. An example of this is shown on the left of Fig. 1 for the nurse rostering problem. It creates the traditional decision variables (line 8), an MDD propagator (line 9), and multiple `amongMDD` constraints for various shifts of windows of length $N1$, $N2$, and $N3$. A call to `x.sub(a,b)` returns the sub-sequence of variables $[x_a, x_{a+1}, ..., x_b]$. Finally, it posts the MDD representing the conjunction of these constraints on line 18. The search is omitted for brevity's sake. The remainder of the paper explains this MDD specification language, its semantics, and how to mechanically derive propagators. Section 2 formally describes MDDs in terms of labeled transition systems. Section 3 introduces the different elements of our description language using states and transitions. We formally introduce the resulting MDD language in Section 4. Section 5 describes the implementation details of HADDOCK, followed by an experimental evaluation Section 6. We conclude in Section 7.

2 Decision Diagrams as Labeled Transition Systems

MDDs are commonly defined as layered directed acyclic graphs [30,5]. For our purposes, however, it is more convenient to formalize an MDD using a labeled

transition system (LTS) as abstraction. Namely, an LTS allows us to describe the *rules* that govern an MDD rather than the MDD itself, and it provides a computational device to compile MDDs. Furthermore, an LTS abstraction can more clearly express the steps of generic MDD compilation, such as the computation of intermediate states, than the concrete acyclic directed graph.

We first recall the definition of labeled transition systems [23]:

Definition 1. *A* labeled transition system *is a triplet $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ where $\mathcal{S}$ is a set of states, $\rightarrow$ is a relation of labeled transitions between states from $\mathcal{S}$ and Λ is a set of labels used to tag transitions.*

A transition from state S_0 to state S_1 (both in $\mathcal{S}$) belonging to the relation $\rightarrow$ is denoted $S_0 \xrightarrow{\ell} S_1$ with $\ell \in \Lambda$. A start state $S_\perp$ has no predecessors according to the transition relation $\rightarrow$ while an end state $S_\top$ has no successors.

Next we define an LTS to represent a single constraint (without loss of generality) from which we will derive our MDD definition.

Definition 2. *Given a constraint $c(x)$ of arity k over an ordered set of variables $x = x_0, \ldots, x_{k-1}$ with domains $D(x_0), \ldots, D(x_{k-1})$, we define the associated labeled transition system as $L(c, x) = \langle \mathcal{S}, \rightarrow, \Lambda \rangle$ in which $\mathcal{S}$, Λ and $\rightarrow$ are defined as follows:*

- *the state set $\mathcal{S}$ is stratified in $k+1$ layers $\mathcal{L}_0$ through $\mathcal{L}_k$ with transitions from $\rightarrow$ connecting states between layers i and $i+1$ exclusively;*
- *the transition label set Λ is defined as $\bigcup_{i \in 0..k-1} D(x_i)$;*
- *a transition between two states $a \in \mathcal{L}_i$ and $b \in \mathcal{L}_{i+1}$ carries a label $v \in D(x_i)$;*
- *the layer $\mathcal{L}_0$ consists of a single* source *state $S_\perp$;*
- *the layer $\mathcal{L}_k$ consists of a single* sink *state $S_\top$.*

By interpreting states as nodes and transitions as arcs, $L(c, x)$ represents a directed acyclic graph. In particular, a path from $S_\perp$ to $S_\top$ corresponds to a complete variable assignment. While Definition 2 directly links variable assignments to transitions between layers in the LTS, the constraint c is implicitly represented in the states $\mathcal{S}$ and the transition function. We next define MDDs as labeled transition systems with specific properties:

Definition 3. *Given a constraint $c(x)$ over an ordered set of variables x, a* relaxed MDD *with respect to $c(x)$ is an LTS $L(c, x)$ such that each state in $\mathcal{S}$ lies on at least one $S_\perp$-$S_\top$ path and each feasible solution to c corresponds to an $S_\perp$-$S_\top$ path. An* exact MDD *with respect to $c(x)$ is a relaxed MDD in which additionally every path from $S_\perp$ to $S_\top$ corresponds to a feasible solution to c.*

In order to compile an MDD, constraints must be appropriately defined in terms of states $\mathcal{S}$ and the transition function $\rightarrow$. We must furthermore specify a suitable state relaxation function to merge non-identical states.

Example 1. Consider the constraint $\textsc{Among}(\{x_0, x_1, x_2, x_3\}, l = 1, u = 2, S = \{1\})$ where each variable has domain $\{0, 1\}$. Similar to [22], we define a state s by four properties, as a tuple $(L^\downarrow(s), U^\downarrow(s), L^\uparrow(s), U^\uparrow(s))$, where

$L^{\downarrow}(s)$ represents the minimum occurrence of values in S along any $S_{\bot}$-s path,
$U^{\downarrow}(s)$ represents the maximum occurrence of values in S along any $S_{\bot}$-s path,
$L^{\uparrow}(s)$ represents the minimum occurrence of values in S along any s-$S_{\top}$ path,
$U^{\uparrow}(s)$ represents the maximum occurrence of values in S along any s-$S_{\top}$ path.

We initialize layer $\mathcal{L}_0$ with the state $S_{\bot} = (0, 0, \infty, \infty)$ and layer $\mathcal{L}_n$ with the state $S_{\top} = (\infty, \infty, 0, 0)$.

We next consider the transition relation $s \xrightarrow{v} s'$ from state $s \in \mathcal{L}_i$ via an arc labeled with $v \in D(x_i)$ to a state $s' \in \mathcal{L}_{i+1}$. This transition affects the properties $L^{\downarrow}(s)$ and $U^{\downarrow}(s)$ as

$$L^{\downarrow}(s') = L^{\downarrow}(s) + (v \in S)$$
$$U^{\downarrow}(s') = U^{\downarrow}(s) + (v \in S).$$

We call such properties *forward* properties since they follow the orientation of the transition. The properties $L^{\uparrow}(s)$ and $U^{\uparrow}(s)$, however, are updated reversely along the transition:

$$L^{\uparrow}(s) = L^{\uparrow}(s') + (v \in S)$$
$$U^{\uparrow}(s) = U^{\uparrow}(s') + (v \in S).$$

We therefore call such properties *reverse* properties. Lastly, we use the state properties to define an existence rule, i.e., transition $s \xrightarrow{v} s'$ exists if

$$(L^{\downarrow}(s) + (v \in S) + L^{\uparrow}(s') \leq u) \wedge (U^{\downarrow}(s) + (v \in S) + U^{\uparrow}(s') \geq l).$$

Fig. 2(a) depicts an LTS that represents an exact MDD for our constraint. The values of the properties are indicated inside the nodes representing the states. One can inspect that each path from $S_{\bot}$ to $S_{\top}$ corresponds to a solution to the constraint, and vice-versa. The width of the exact MDD is three.

To define a relaxed MDD, we specify a *merge* operator that takes two states $a, b \in \mathcal{L}_i$ and merges them into a single state s. For Among, we define it as:

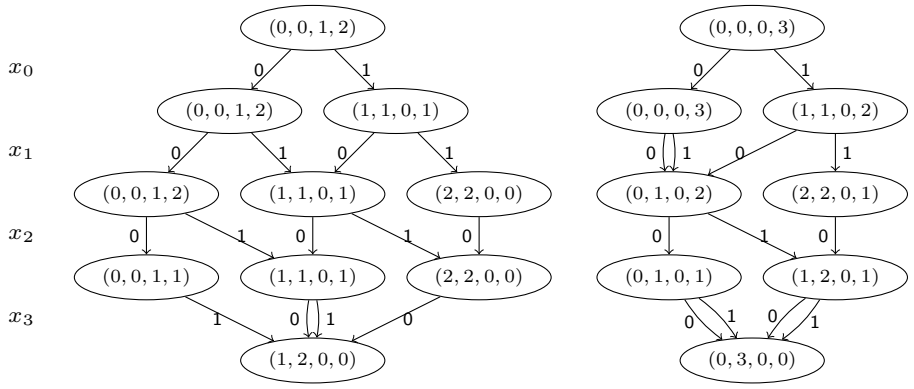
$$L^{\downarrow}(s) = \min(L^{\downarrow}(a), L^{\downarrow}(b)) \qquad L^{\uparrow}(s) = \min(L^{\uparrow}(a), L^{\uparrow}(b))$$
$$U^{\downarrow}(s) = \max(U^{\downarrow}(a), U^{\downarrow}(b)) \qquad U^{\uparrow}(s) = \max(U^{\uparrow}(a), U^{\uparrow}(b)).$$

Observe that this merging operator *relaxes* the state computation. It therefore may introduce non-solutions to the MDD but will never lead to the removal of solutions. Fig. 2(b) depicts a relaxed MDD, where the width is limited to a maximum of two. Each solution to the constraint is present as a path from $S_{\bot}$ to $S_{\top}$, but it also contains non-solutions, e.g., $(0, 0, 0, 0)$. □

The properties of a state s in the LTS represent information over the collection of paths from $S_{\bot}$ to s (i.e., prefixes) and from s to $S_{\top}$ (i.e., suffixes). Note that producing a suitable LTS specification for any given constraint c will yield a different set of state properties, transition definitions, existence conditions, and merging rules. The following sections explain how to do this systematically.

3 States and Transition Functions

This section describes the core elements of the LTS representation in Haddock in terms of states and transitions.



a. Exact decision diagram b. Relaxed decision diagram

Fig. 2. Exact (a) and relaxed (b) MDDs for the constraint $\textsc{Among}(\{x_0, x_1, x_2, x_3\}, l = 1, u = 2, S = \{1\})$, where all variable domains are $\{0, 1\}$. Each state s depicts its tuple of properties $(L^{\downarrow}(s), U^{\downarrow}(s), L^{\uparrow}(s), U^{\uparrow}(s))$ as defined in Example 1.

3.1 States

For exposition purposes, we assume that states are defined by properties that are integer-valued, but we note that these can represent Booleans and even richer types such as floating points or sets.

Definition 4. *A* state *is a tuple* $s = \langle P_{i_0}, P_{i_1}, \ldots, P_{i_{n-1}} \rangle$ *with* n *properties denoted* P_{i_k} *where each* $P_{i_k} \in \mathbb{Z}$.

It is important to realize that our LTS, and resulting MDD, can carry properties for multiple constraints. The identifiers (names) created for a constraint c are collected in a set $\mathcal{P}(c) = \{i_0, \ldots, i_{n-1}\}$. These identifiers are unique across constraints, i.e., two constraints are guaranteed to use different property names.

3.2 Forward and Reverse Transition Rules

As shown in Example 1, a transition $s \xrightarrow{\ell} s'$ can be processed following the orientation of the transition (forward), or reversely. A state property may be subject to transition rules that are forward, reverse, or both. We therefore define the set of property indices $\Delta^{\downarrow} \subseteq \{0, \ldots, n-1\}$ and $\Delta^{\uparrow} \subseteq \{0, \ldots, n-1\}$ to indicate the properties that will be processed by the forward and reverse transition rules, respectively. We start by defining how individual properties are updated by a transition, by means of forward and reverse *property transition rules*:

Definition 5. *A* forward property transition rule *for property* P_i *is a function* $T_i^{\downarrow} : \mathcal{S} \times \mathbb{Z} \times \mathcal{D} \to \mathbb{Z}$. *It takes as input a source state* $s \in \mathcal{L}_j$, *the layer index* j *(i.e., variable* x_j*), and a domain value* $v \in D(x_j)$ *to produce a value for property* P_i *in the destination state* $s' \in \mathcal{L}_{j+1}$.

Definition 6. *A* reverse property transition rule *for property P_i is a function $T_i^{\uparrow} : \mathcal{S} \times \mathbb{Z} \times \mathcal{D} \to \mathbb{Z}$. It takes as input a state $s \in \mathcal{L}_{j+1}$, the layer index j (i.e., variable x_j), and a domain value $v \in D(x_j)$ to produce a value for property P_i in the destination state $s' \in \mathcal{L}_j$.*

We next use the individual property transition rules to define forward and reverse *state transition rules*:

Definition 7. *A* forward state transition rule *is a function $T^{\downarrow} : \mathcal{S} \times \mathcal{S} \times \mathbb{Z} \times \mathcal{D} \to \mathcal{S}$. Given a source state $s = \langle P_0, \ldots, P_{n-1} \rangle \in \mathcal{L}_j$, a destination state $d = \langle Q_0, \ldots, Q_{n-1} \rangle \in \mathcal{L}_{j+1}$, the layer index j (i.e., variable x_j), a domain value $v \in D(x_j)$ and the set of forward property indices $\Delta^{\downarrow}$, it computes the forward property transitions of the successor state s' in $\mathcal{L}_j$. The function is defined as follows:*

$$\forall i \in \{0, \ldots, n-1\} \;\; : \;\; Q'_i = \begin{cases} T_i^{\downarrow}(s, j, v) & \text{if } \; i \in \Delta^{\downarrow} \\ Q_i & \text{if } \; i \notin \Delta^{\downarrow} \end{cases}$$

and, finally

$$T^{\downarrow}(s, d, j, v) = \langle Q'_0, \ldots, Q'_{n-1} \rangle.$$

Definition 8. *A* reverse state transition rule *is a function $T^{\uparrow} : \mathcal{S} \times \mathcal{S} \times \mathbb{Z} \times \mathcal{D} \to \mathcal{S}$. Given a source state $s = \langle P_0, \ldots, P_{n-1} \rangle \in \mathcal{L}_{j+1}$, a destination state $d = \langle Q_0, \ldots, Q_{n-1} \rangle \in \mathcal{L}_j$, the layer index j (i.e., variable x_j), a domain value $v \in D(x_j)$ and the set of reverse property indices $\Delta^{\uparrow}$, it computes the reverse property transitions of the successor state s' in $\mathcal{L}_j$. The function is defined as follows:*

$$\forall i \in \{0, \ldots, n-1\} \;\; : \;\; Q'_i = \begin{cases} T_i^{\uparrow}(s, j, v) & \text{if } \; i \in \Delta^{\uparrow} \\ Q_i & \text{if } \; i \notin \Delta^{\uparrow} \end{cases}$$

and, finally

$$T^{\uparrow}(s, d, j, v) = \langle Q'_0, \ldots, Q'_{n-1} \rangle.$$

In the definitions of the state transition rules, the destination state d and the source state s provide values for properties not listed in $\Delta^{\downarrow}$ and $\Delta^{\uparrow}$, respectively.

3.3 Relaxation Functions

By design, a state s is generally subject to multiple forward and reverse state transitions. These are aggregated, or merged, via pointwise *property relaxation functions*. Relaxation functions are also applied to merge non-identical states when a layer exceeds the given maximum MDD width.

Definition 9. *A* property relaxation function *for property P_i takes the form $R_i : \mathcal{S} \times \mathcal{S} \to \mathbb{Z}$. It takes as input states $s^{\ell} = \langle P_0^{\ell}, \ldots, P_{n-1}^{\ell} \rangle$ and $s^r = \langle P_0^r, \ldots, P_{n-1}^r \rangle$ and returns $P'_i = R_i(s^{\ell}, s^r)$ as the merged property.*

The individual property relaxations are used to define *state relaxation functions*:

Definition 10. *A* state relaxation function *takes the form $R : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$. Given states s^{ℓ} and s^r it computes $\langle R_0(s^{\ell}, s^r), \ldots, R_{n-1}(s^{\ell}, s^r) \rangle$.*

3.4 Transition Existence Functions

A critical component of MDD propagation is *arc filtering*, i.e., the removal of arcs that do not belong to any path corresponding to a solution to the constraint. Arc filtering is performed by applying *transition existence functions*, which rely on the source and destination states, as well as the transition label:

Definition 11. *A* transition existence function *takes the form* $E_t : \mathcal{S} \times \mathcal{S} \times \mathbb{Z} \times \mathcal{D} \to \mathbb{B}$. *Given a parent state* p, *a child state* c, *a layer* j *for state* c, *and a transition label* v, *the function returns the existence of path* $S_\top \rightsquigarrow p \xrightarrow{v} c \rightsquigarrow S_\bot$ *whose labels correspond to a solution to the constraint.*

3.5 State Functions

State transitions are defined *pointwise* in terms of forward and reverse property transitions. This does not provide for state-wide reasoning involving multiple state properties. In some cases, for example the MDD propagation for Sequence constraints [4], it is desirable to process multiple state properties simultaneously. *State update functions* provide this capability:

Definition 12. *A* state update function *takes the form* $U : \mathcal{S} \to \mathcal{S}$. *It is a state transformation function that updates property* P_i *based on properties* $P_0, \ldots, P_{i-1}, P_{i+1}, \ldots, P_{n-1}$ *for* $i = 0, \ldots, n-1$.

Analogous to transition existence functions, we also define *state existence functions*:

Definition 13. *A* state existence function *takes the form* $E_s : \mathcal{S} \to \mathbb{B}$. *Given a state* p *the function returns the existence of paths* $S_\top \rightsquigarrow p$ *and* $p \rightsquigarrow S_\bot$ *whose labels correspond to a solution to the constraint.*

Example 2. Continuing Example 1, we recognize that the state properties, the forward and reverse transition functions, the relaxation functions, and the transition existence function all follow the specifications in this section. As an example of a state existence function for Among, we could define for a state s:

$$E_s(s) = (L^{\downarrow}(s) + L^{\uparrow}(s) \leq u) \wedge (U^{\downarrow}(s) + U^{\uparrow}(s) \geq l).$$

□

4 MDD Language

It is, perhaps, valuable to cast an LTS as a *virtual machine formalism* in which to capture the computational aspects of an MDD. An LTS faithfully models state derivation via tentative variable assignments as well as state relaxations with transitions. Yet, it remains a computational artifact that is delicate to describe and harder to maintain. This section introduces an *MDD language* to express the set of source-to-sink paths forming the MDD that an LTS computes. To a

large extent, MDD languages are to LTS what regular expressions are to DFAs. Namely, an MDD language enables a developer to elevate the discourse and capture the language to be recognized in a fairly compact and declarative fashion. It leaves to a "compiler" the delicate task of producing the state machine. Figure 1 provides an illustration of an MDD language given in pseudo-C++. The purpose of this section is to describe the language more precisely.

Definition 14. *Given a constraint $c(x_0, \ldots, x_{k-1})$ over an ordered set of variables $X = \{x_0, \ldots, x_{k-1}\}$ with domains $D(x_0), \ldots, D(x_{k-1})$ the* MDD language *for c is a tuple $\mathcal{M}_c = \langle X, \mathcal{P}, S_\bot, S_\top, T^\downarrow, T^\uparrow, U, E_t, E_s, R\rangle$ where $\mathcal{P}$ is the set of properties used to model states, $S_\bot$ is the source state, $S_\top$ is the sink state, $T^\downarrow$ is the forward state transition rule, $T^\uparrow$ is the reverse state transition rule, U is the state update function, E_t is the transition existence function, and E_s is the state existence function.*

We note that with the exception of X, all elements in the tuple $\mathcal{M}_c$ depend on the constraint c.

An MDD language is an abstraction to describe the states and rules that define a labeled transition system $\langle \mathcal{S}, \rightarrow, \Lambda\rangle$ for a given constraint. The LTS can be interpreted as a computational device that recognizes acceptable paths forming the MDD for the constraint. This is formalized in the following theorem.

Theorem 1. *Let $c(x_0, \ldots, x_{k-1})$ be a constraint over an ordered set of variables $X = \{x_0, \ldots, x_{k-1}\}$ with domains $D(x_0), \ldots, D(x_{k-1})$. An MDD language $\mathcal{M}_c$ is sufficient to define an exact or relaxed MDD for c.*

Proof. The LTS defines states by the properties in $\mathcal{P}$, and relies on $S_\bot$ and $S_\top$ for initialization. It produces new states by using the forward and reverse state transition rules $T^\downarrow, T^\uparrow$, the state update rule U, and the values in $D(x_j)$ for layer j. The existence of each transition and state is given by E_t and E_s.

We use the LTS to compute an MDD by organizing the transitions into $k+1$ layers where $S_\bot$ initializes layer 0, $S_\top$ initializes layer k, and the transition values for layer j correspond to the domain values of x_j. All transitions out of states in layer $k-1$ are directed to $S_\top$ via the relaxation operator R. This process may create paths not connected to $S_\bot$ or $S_\top$; we remove such paths from the LTS.

If no maximum width per layer is imposed, the process will create an exact MDD, which follows from the definition of the properties, state transition rules, state update rules, and transition and state existence functions, i.e., each path from $S_\bot$ to $S_\top$ corresponds to a solution to c.

In presence of a maximum width per layer, we can arbitrarily merge nodes using the relaxation operator R, which by definition computes a relaxation of the merged states. Hence this process produces a relaxed MDD. □

Since we use MDDs to represent and propagate multiple constraints simultaneously, we need a formalism that supports this functionality. Indeed, MDD languages can be combined by defining an appropriate conjunction operator, and the result is again an MDD language, as shown in the following theorem:

Theorem 2. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be MDD languages. There exists a conjunction operator $\wedge$ for MDD languages such that $\mathcal{M}_1 \wedge \mathcal{M}_2$ is also an MDD language, and represents the set of paths that are common to $\mathcal{M}_1$ and $\mathcal{M}_2$.*

The proof relies on the definition of the conjunction operator, which is detailed in Appendix [5]. It essentially concatenates the tuples that define $\mathcal{M}_1$ and $\mathcal{M}_2$. For example, if $\mathcal{M}_1$ and $\mathcal{M}_2$ are defined on overlapping sets of variables X_1 and X_2, the conjunction is defined on $X_1 \cup X_2$ (e.g., ordered first by X_1 and then by X_2). The conjunction operator forms the basis of the implementation in HADDOCK for composing multiple constraints into an MDD.

Observe that Theorem 1 leaves open many design choices for an actual implementation of the LTS and associated MDD. Section 5 explains how, operationally, HADDOCK uses MDD languages to produce the actual LTS and MDD.

5 Implementation

The purpose of an MDD specification is to mechanize the construction and the propagation of an actual MDD. Sub-section 5.1 first considers the creation (posting) of an MDD. Sub-section 5.2 explores the actual propagation of *events* occurring through the course of a CP-style search.

5.1 Posting

HADDOCK embraces an incremental refinement scheme to construct an MDD [17]. It starts by creating a width-one MDD connecting $S_\bot$ to $S_\top$ which is then refined via *node splitting* until each layer $\mathcal{L}_i$ for $i \in 1..k-1$ reaches at most its target width w and the MDD settles into an MDD propagation fixpoint.

Initialization Given an MDD language $\langle X, \mathcal{P}, S_\bot, S_\top, T^\downarrow, T^\uparrow, U, E_t, E_s, R\rangle$, the source state $S_\bot$ and sink state $S_\top$ have their properties initialized to values suitable for an empty prefix and suffix, respectively, for the constraint c the MDD models. Specifically, each property in $P_i \in \mathcal{P}$ subject to a forward property transition rule $T_i^\downarrow$ should be initialized in $S_\bot$ while each property P_i subject to a reverse property transition rule $T_i^\uparrow$ should be initialized in $S_\top$.

Example 3. Recall that for AMONG, $\mathcal{P} = \{L^\downarrow, U^\downarrow, L^\uparrow, U^\uparrow\}$, and, therefore, let $S_\bot = [L^\downarrow \mapsto 0, U^\downarrow \mapsto 0]$ and $S_\top = [L^\uparrow \mapsto 0, U^\uparrow \mapsto 0]$. Namely, the forward properties are initialized to 0 in the source state and the reverse properties are initialized to 0 in the sink state. □

Construction The construction of an MDD proceeds in phases. First, a two pass (forward and reverse) process creates a width one MDD. The second phase *widens* the MDD up to width w through node splitting.

[5] Curious readers can find it in the online supplement.

First phase The forward pass loops through layers $1..k-1$ to generate states. Assume that the algorithm operates on layer $\mathcal{L}_j$, i.e., all layers $\mathcal{L}_0 \cdots \mathcal{L}_{j-1}$ are already constructed (each layer is a singleton in this phase). It picks the one state $s \in \mathcal{L}_{j-1}$, and, for each value $v \in D(x_{j-1})$, it computes the successor state s' according to $s \xrightarrow{v} s'$, i.e., it evaluates

$$s_v = T^{\downarrow}(s, S_\top, j, v).$$

Note how it relies on $S_\top$ as an initial approximation for the reverse-defined properties. Finally, it uses the state update function U, the relaxation function R and the domain of $D(x_{j-1}) = \{v_0, v_1, v_2, \ldots, v_\ell\}$ to deliver

$$s' = U(R(\cdots R(R(s_{v_0}, s_{v_1}), s_{v_2}) \cdots, s_{v_\ell}))$$

and define $\mathcal{L}_j = \{s'\}$. The reverse pass iterates backward through the layers. At iteration j, it uses the singleton state $s' \in \mathcal{L}_j$, its one child $c \in \mathcal{L}_{j+1}$, and a value $v \in D(x_j)$ to compute $s_v = T^{\uparrow}(c, s', j, v)$. The computation integrates the down state s' obtained from the forward pass. Once again, an application of the relaxation yields the now final value for s' via

$$s' = U(R(\cdots R(R(s_{v_0}, s_{v_1}), s_{v_2}) \cdots, s_{v_\ell})) \text{ with } \mathcal{L}_j = \{s'\}.$$

Second phase The purpose is to widen each layer of the MDD to its final width. The *splitting* algorithm is applied to relaxed nodes in a layer $\mathcal{L}_j$ for which $|\mathcal{L}_j| < w$. Let s be such a state and let $\delta^-(s)$ denote the set of its parent states in layer $\mathcal{L}_{j-1}$ and $\delta^+(s)$ denote the set of its child states in layer $\mathcal{L}_{j+1}$. One can compute, for each existing transition $p \xrightarrow{v} s$ with $p \in \delta^-(s)$, the true state one would reach from p according to value v, i.e.,

$$s_p = T^{\downarrow}(p, s, j-1, v) \;\; : \forall p \in \delta^-(s) \;\; \text{and } v \text{ on } p \xrightarrow{v} s.$$

Several parents p might yield the same s_p, which conveys that multiple transitions lead to the same refined state. One can compute

$$\mathit{refine}(s) = \{s_p \mid \forall p \in \delta^-(s) \;\; \text{and } v \text{ on } p \xrightarrow{v} s\}$$

as the set of refined states meant to replace s and endow each $s' \in \mathit{refine}(s)$ with a transition from its parents and the same child states as s, i.e, $\delta^+(s)$. Some transitions to states in $\delta^+(s)$ may be invalid according to E_t and are not added. Any element of $\mathit{refine}(s)$ that ends up childless as a result must be deleted. Such deletion can trickle back up through multiple layers and bring their width below the desired value. In such a situation one may wish to *reboot* the splitting to an earlier layer rather than continuing on the current one. Pragmatically, it may be useful to bound how far one can "reboot", which we investigate in Section 6.

It is desirable to order $\mathit{refine}(s)$ as its cardinality, together with that of $\mathcal{L}_j \backslash \{s\}$ may exceed w in which case not all states in $\mathit{refine}(s)$ can be adopted "as is" and some merging is required. Namely, given a state ordering relation $\preccurlyeq$ between

states, the ordering of $\mathit{refine}(s)$ is $s'_0 \preccurlyeq s'_1 \preccurlyeq \ldots \preccurlyeq s'_{m-1}$ ($|\mathit{refine}(s)| = m$). This ordering can induce up to ℓ equivalence classes of states deemed similar enough to warrant the use of a single state to represent each class. Naturally, $\ell = w - |\mathcal{L}_j| + 1$, i.e., ℓ is an upper bound on the number of classes based on the number of *unused slots* in layer j. The new layer $\mathcal{L}_j$ becomes[6]

$$\mathcal{L}'_j = \mathcal{L}_j \setminus \{s\} \cup \left(\bigcup_{C_k \in \{C_0, \ldots, C_{l-1}\}} U(R(C_k)) \right)$$

with $C_0 \cdots C_{l-1}$ as the $l \leq \ell$ classes formed from $s'_0 \preccurlyeq \ldots \preccurlyeq s'_{m-1}$. Layer j's width satisfies $|\mathcal{L}'_j| \leq w$. Changes to the topology of the MDD (state addition, deletion and transitions) are *events* that require a fixpoint propagation to update the values of properties held in the affected states, which is discussed next.

5.2 Propagation

MDD Events Propagation occurs in response to *events* affecting the MDD or its connection to finite domain variables. Solver events like $\mathit{del}(v, x_j)$ reporting the loss of a value v, i.e., $v \notin D(x_j)$ for a variable x_j appearing in the MDD induce MDD events to convey the deletion of all transitions between layers j and $j+1$ labeled by v. MDD events are handled *within* the MDD and are detailed below:

$\mathit{del}(s)$: loss of state s in a layer
$\mathit{add}(s)$: addition of state s in a layer
$\mathit{del}(p \xrightarrow{v} c)$: loss of a transition from p to c
$\mathit{add}(p \xrightarrow{v} c)$: addition of a transition from p to c
$\mathit{state}(i, s)$: properties $i \subseteq \mathcal{P}(c)$ appearing in state s have changed

The first four events change the topology of the MDD and affect one or more states. The loss of $p \xrightarrow{v} c$ means that the properties of p and c may now be outdated and should be refreshed as that transition might have contributed to a relaxation. For any event e, let $\mathit{affected}^{\downarrow}(e)$ and $\mathit{affected}^{\uparrow}(e)$ be the set of states below and above the state, respectively, or the transition in e and affected by it.

Overall propagation Any state s of an MDD depends on its parent states $\delta^-(s)$ and its child states $\delta^+(s)$. The layered and acyclic structure of the MDD graph provides a natural strategy for updating states when changes occur. Since the MDD is Berge acyclic [3], processing the updates through forward and reverse passes that consider changes in reverse topological order and topological order, respectively, is sensible. Passes can be repeated until a fixpoint is reached.

To this end, one needs an event list E and two priority queues $Q^{\downarrow}$ and $Q^{\uparrow}$ to hold onto states. The priority is simply the layer index of the state. Processing an event $e \in E$ schedules $\mathit{affected}^{\downarrow}(e)$ in $Q^{\downarrow}$ and $\mathit{affected}^{\uparrow}(e)$ in $Q^{\uparrow}$ accordingly. Eventually, one processes the states in $Q^{\uparrow}$ in reverse priority order, followed by

[6] We take some liberty with notation and refer to the relaxation R over a set of states.

the states in $Q^{\downarrow}$ in priority order. This propagation may delete transitions as well as states, which induces additional rounds of splitting (as described in the previous sub-section). Pragmatically, it may be desirable to *bound* the number of passes in which splitting occurs to curtail the computational efforts.

Event Propagation A comprehensive treatment of events is not possible within page limitations. Consider as an example the propagation rule for the event $e = state(i, s)$ concerning a change to the properties in i for state s in layer $\mathcal{L}_j$. Processing e means scheduling $\delta^-(s)$ in $Q^{\uparrow}$ and $\delta^+(s)$ in $Q^{\downarrow}$. When a child c of s is pulled from $Q^{\downarrow}$ (the case with $Q^{\uparrow}$ is similar), state c must be updated since s has changed and the relation $s \xrightarrow{v} c$ connects them. This can be done with

$$c_p = T^{\downarrow}(p, c, j, v_p) \ \forall p \in \delta^-(c), \text{with } p \xrightarrow{v_p} c$$

and computing the relaxation of states c_p for all parents p to yield

$$c' = U(R(\{c_p | p \in \delta^-(c), \text{with } p \xrightarrow{v_p} c\})).$$

If $E_s(c')$ does not hold, c' itself is no longer sound and is deleted. If $E_s(c')$ holds and $c' \neq c$, c is changing. Therefore, every transition $c \xrightarrow{v} d$ must be tested with E_t and invalid transitions removed. Such removal should trigger the scheduling of affected nodes. Since c is now changed to c', an event $state(\psi, c')$ should be processed (ψ refers to the subset of properties whose value differ in c and c').

6 Empirical Evaluation

HADDOCK is part of `MiniC++`, a C++ implementation of the MiniCP specification [26]. All benchmarks were executed on a Macbook Pro with an i7-5557U processor and 16GB. While a generic implementation is unlikely to match dedicated implementations, HADDOCK let developers produce MDD-based propagators for global constraints with minimal effort. We demonstrate below 1) how HADDOCK compares to an existing MDD implementation, 2) its new modeling capabilities, and 3) the performance impact of the reboot depth parameter.

Experiment 1: Comparison to State-of-the-Art. First, we compare HADDOCK to the 'Dedicated' AMONG MDD propagator developed in [22,21] and a classic finite-domain model (written in MiniCP) that uses a cumulative-sums encoding for SEQUENCE constraints as a reasonable baseline for domain propagation. We evaluate these methods on the nurse rostering benchmark problems from [22]. Those problems ask to schedule work shifts for a nurse, subject to a collection of AMONG (or SEQUENCE) constraints. There are three classes of instances with different lower and upper bounds on the number of work days in a sequence. Class C-I requires at most 6 out of 8 consecutive work days and at least 22 out of 30 consecutive work days. C-II uses 6 out of 9 and 20 out of 30 while C-III uses 7 out of 9 and 22 out of 30. Each instance also requires 4 or 5 work days each week and

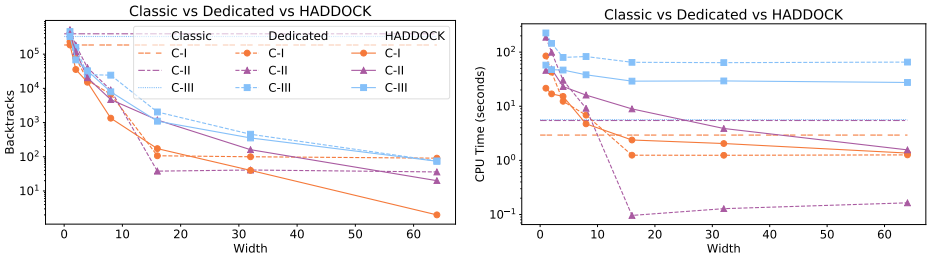


Fig. 3. Backtracks (left) and CPU time (right) for finding all solutions for `amongNurse`.

uses a horizon of 40 days. Runtimes reported in [21] were scaled to account for processor differences. We used the highest Geekbench 5 scores reported for our machine CPU (825) and an average score of 330 (standard deviation 10) for an Intel Xeon E5345 that matches the 2.3GHz characterization in [21]. This yields a scaling by $\frac{330}{825} = 0.4$. Important differences between HADDOCK and [21] are that the implementation of [21] does not compute an MDD propagation fixpoint and adopts a dedicated splitting heuristic that differs from ours.

Figure 3 compares the search tree size (left) and runtimes (right) between the three approaches for finding *all solutions* (each model uses the same lexicographic search strategy). 'Classic' refers to the conventional finite-domain model, 'Dedicated' refers to [21], and 'Haddock' is our implementation with a reboot depth of 0 for clearer comparison. Observe that HADDOCK starts, at width 1, with the exact same number of backtracks as Classic (a bit better than Dedicated, which is attributed to the lack of a fixpoint within [21]). Second, the number of backtracks of HADDOCK are roughly comparable to [21]. Third, the runtimes are within a small factor of each other with HADDOCK in front for class C-III and behind (by a factor in the 1x-10x range) on C-II. Those results are very promising given the generic nature of HADDOCK.

Experiment 2: Modeling Capabilities We next demonstrate the modeling capabilities of HADDOCK on the classic *All-Interval Series* problem (#007 on CSPLIB). Given an integer n, the problem asks to find a vector $x = (x_0, \ldots, x_{n-1})$ such that x is a permutation of $\{0, \ldots, n-1\}$, and the interval vector $y = (|x_1 - x_0|, |x_2 - x_1|, \ldots, |x_{n-1} - x_{n-2}|)$ is a permutation of $\{1, \ldots, n-1\}$.

To model this problem using MDDs, we defined the ABSDIFFMDD(z_0, z_1, z_2) constraint in HADDOCK, representing the relation $|z_0 - z_1| = z_2$. It maintains the set of values taken by each of the variables and defines explicit arc existence functions based on this relationship. We then introduce constraints ABSDIFFMDD(x_i, x_{i+1}, y_i), for $i = 0, \ldots, n-2$, which gives us a natural variable ordering, interleaving vectors x and y. Lastly, we define ALLDIFFMDD(x) and ALLDIFFMDD(y) constraints to model the permutations. The entire ABSDIFFMDD language requires less than 120 lines for declaring its states, transitions, and relaxation rules. Likewise, the ALLDIFFMDD language requires only 70 lines. For these constraints, HADDOCK applies a generic state ordering based on the number of parent states (smallest first) during the node splitting process.

Reboot	Width 1	Width 2	Width 4	Width 8	Width 16	Width 32	Width 64
0	10,062	9,459	7,815	6,196	4,027	2,368	1,511
2	10,062	9,332	7,522	6,191	3,796	2,264	1,470
4	10,062	9,069	6,972	5,279	3,282	2,100	1,357
8	10,062	5,698	3,277	2,283	1,626	1,839	316
MAX	10,062	1,155	140	40	34	23	6

Table 1. MDD propagation on the All-Interval Series problem ($n = 11$).

We apply a lexicographic search on the x variables to compare the performance over different settings. We report results for $n = 11$ (finding all solutions) in Table 1. To evaluate the impact of MDD propagation, consider the base case in which the reboot depth is set to 0. The table shows how increasing the width from 1 (domain propagation) to larger widths reduces the search tree size, yielding a factor 10 for maximum width 64. It is encouraging to see how MDDs can be used like any other global constraints within a CP solver and deliver huge reductions in the size of the search tree. We do note, however, that the reduction in search tree size has no positive impact on the solving time. It remains an area for future work to exploit better split ordering and an optimized implementation.

Experiment 3: Reboot Depth Sensitivity. We next investigate the impact of varying the reboot depth. Consider Table 1 once again. Rows 2 through 5 convey the impact of reboots on the search tree size at all considered widths. Recall that HADDOCK only relies on a syntactic value (the number of parents) to rank states. Also recall that rebooting considers abandoning splitting at layer k when it has induced state deletion in prior layers and return to those. The `reboot` parameter controls how far back HADDOCK can jump when returning to earlier layers. In this situation, rebooting can have a *dramatic* impact on the size of the search at any width. The most extreme parameter settings of width=64 and unbounded reboots yield a search tree with only 6 nodes, i.e., 4 orders of magnitude compared to the baseline in the upper left corner of the table.

7 Conclusion

This paper introduced HADDOCK, a system for Handling Automatically Decision Diagrams over Constraints Kernels. It described the language and framed its implementation in terms of a compilation down to an LTS form. The resulting system is generic and capable of capturing and automatically deriving implementation for several MDDs that were proposed before and for which only dedicated implementations exist. The empirical evaluation showed that this very first implementation of a generic MDD engine exhibits search tree reductions that are qualitatively on par with prior work and achieves comparable runtime.

Acknowledgments Willem-Jan van Hoeve was partially supported by ONR Grant No. N00014-18-1-2129 and NSF Award #1918102. L. Michel and R. Gentzel were partially supported by Comcast under Grant #15228, Synchrony under #790057267 and ONR/NIUVT under #N000014-20-1-2040.

References

1. S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.
2. H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.
3. Claude Berge. *Hypergraphs - combinatorics of finite sets.*, volume 45 of *North-Holland mathematical library*. North-Holland, 1989.
4. D. Bergman, A. A. Cire, and W.-J. van Hoeve. MDD Propagation for Sequence Constraints. *JAIR*, 50:697–722, 2014.
5. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Springer, 2016.
6. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Discrete Optimization with Decision Diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
8. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293– 318, 1992.
9. K. C. K. Cheng, W. Xia, and R. H. C. Yap. Space-time tradeoffs for the regular constraint. In *Constraint Programming Proceedings (CP 2012)*, volume 7514 of *LNCS*, pages 223–237. Springer, 2012.
10. K. C. K. Cheng and R. H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In G. Brewka et al., editor, *Proceedings of ECAI*, pages 78–82. IOS Press, 2006.
11. A. A. Cire and W.-J. van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61(6):1411–1428, 2013.
12. Diego de Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. Compiling CP subproblems to mdds and d-dnnfs. *Constraints*, 24(1):56–93, 2019.
13. G. Gange, V. Lagoon, and P. J. Stuckey. Fast Set Bounds Propagation using BDDs. In M. Ghallab et al., editor, *Proceedings of ECAI*, pages 505–509. IOS Press, 2008.
14. Graeme Gange, Peter J. Stuckey, and Pascal Van Hentenryck. Explaining propagators for edge-valued decision diagrams. In *Proceedings of CP*, volume 8124 of *LNCS*, pages 340–355. Springer, 2013.
15. Graeme Gange, Peter J. Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints*, 16(4):407–429, 2011.
16. T. Hadžić and J. N. Hooker. Cost-bounded binary decision diagrams for 0-1 programming. Technical report, Carnegie Mellon University, 2007.
17. T. Hadžić, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In P. J. Stuckey, editor, *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *Lecture Notes in Computer Science*, pages 448–462. Springer, 2008.
18. T. Hadžić, J. N. Hooker, and P. Tiedemann. Propagating separable equalities in an MDD store. In L. Perron and M. A. Trick, editors, *CPAIOR 2008 Proceedings*, volume 5015 of *Lecture Notes in Computer Science*, pages 318–322. Springer, 2008.
19. T. Hadzic, E. O'Mahony, B. O'Sullivan, and M. Sellmann. Enhanced Inference for the Market Split Problem. In *Proceedings of ICTAI*, pages 716–723. IEEE Computer Society, 2009.
20. P. Hawkins, V. Lagoon, and P.J. Stuckey. Solving Set Constraint Satisfaction Problems Using ROBDDs. *JAIR*, 24(1):109–156, 2005.

21. S. Hoda. *Essays on equilibrium computation, MDD-based constraint programming and scheduling*. PhD thesis, Carnegie Mellon University, 2010.
22. S. Hoda, W.-J. van Hoeve, and J. N. Hooker. A Systematic Approach to MDD-Based Constraint Programming. In *Proceedings of CP*, volume 6308 of *LNCS*, pages 266–280. Springer, 2010.
23. R. M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, 1976.
24. J. Kinable, A. A. Cire, and W.-J. van Hoeve. Hybrid Optimization Methods for Time-Dependent Sequencing Problems. *European Journal of Operational Research*, 259(3):887–897, 2017.
25. V. Lagoon and P. J. Stuckey. Set domain propagation using ROBDDs. In M. Wallace, editor, *Proceedings of CP*, volume 3258 of *LNCS*, pages 347–361, 2004.
26. Laurent Michel, Pierre Schaus, Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming, 2018. Available from `https://minicp.bitbucket.io`.
27. Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.
28. G. Perez and J.-C. Régin. Efficient Operations On MDDs for Building Constraint Programming Models. In *Proceedings of IJCAI*, pages 374–380, 2015.
29. Guillaume Perez and Jean-Charles Régin. Soft and cost MDD propagators. In *Proceedings of AAAI*, pages 3922–3928. AAAI Press, 2017.
30. I. Wegener. *Branching programs and binary decision diagrams: theory and applications*. SIAM monographs on discrete mathematics and applications. Society for Industrial and Applied Mathematics, 2000.